Old Maid Game (Report)

Name: Akram Jaghoub

1.0 Introduction

Welcome to the documentation for the "Maid Game" assignment. This document outlines my implementation and design of the Maid Game, a Java-based multi-player card game. The assignment showcases my skills in object-oriented programming, multi-threading, synchronization, and game logic.

In this documentation, I'll provide an overview of the Maid Game's components, including class interactions and game flow. I'll also highlight the challenges I tackled during development. Through this assignment, I aimed to improve my programming abilities while creating a functional and engaging card game simulation.

I'll cover the game's architecture, class roles, multi-threading handling, and game mechanics. This project allowed me to explore game development concepts and refine my grasp of concurrent programming.

2.0 Object Oriented Design

1. Abstraction

The codebase effectively employs abstraction to ensure a modular and maintainable design. Key classes such as Player, Card, Deck, and PlayerQueue follow the Single Responsibility Principle, each encapsulating specific attributes, and behaviors. Moreover, the abstract class Game and its implementation in MaidGame provide a clear separation of concerns, embodying a template for game structures while enabling specialized game types to be added by extending the abstract class.

2. Inheritance and Polymorphism

The codebase utilizes inheritance to establish a hierarchy of card types, such as standard cards and jokers, inheriting shared attributes and behaviors. Polymorphism then enables these diverse card types to be treated uniformly, ensuring flexibility, and streamlined code logic. This combination enhances adaptability and maintains a coherent structure.

3. Encapsulation

The encapsulation principle is upheld through well-defined access modifiers. Private attributes and methods in classes prevent unauthorized external access only through getters and setters, safeguarding the internal state. For instance, the Card class encapsulates properties like suit, value, and color.

```
5 usages
private final String value;
5 usages
private final Color color;

1 usage   Akram Jaghoub
public Card(String suit, String value, Color color) {
    this.suit = suit;
    this.value = value;
    this.color = color;
}


no usages   Akram Jaghoub
public String getSuit() { return suit; }

no usages   Akram Jaghoub
public String getValue() { return value; }
```

4. Composition

The Player class employs composition to manage player interactions with the game. By encapsulating a Deck instance within the Player class, the code achieves a hierarchical relationship between entities. This approach fosters the creation of more complex objects through composition.

5. Utility and Singleton Classes

The inclusion of utility classes, such as GameUtil, reinforces code organization and reusability which also prevents others from initializing it. By centralizing functionalities related to dealing cards and printing game state, the utility class enhances the codebase's clarity. Moreover, the singleton design pattern graces classes like Deck and PlayerQueue, ensuring single instances and streamlined resource management throughout the application.

```
3 usages
private static PlayerQueue instance;
10 usages
private final Queue<Player> players;
3 usages
private Player currentPlayer;
4 usages
private final Object lock;

1 usage   Akram Jaghoub
private PlayerQueue(){
    players = new LinkedList<>();
    lock = new Object();
}

6 usages   Akram Jaghoub
public static PlayerQueue getInstance(){
    if(instance == null)
        instance = new PlayerQueue();
    return instance;
}
```

```
public class GameUtil {
    no usages   Akram Jaghoub
    private GameUtil() {}


    1 usage   Akram Jaghoub
    public synchronized static void dealCardsToPlayers() {
        PlayerQueue playerQueue = PlayerQueue.getInstance();
        Deck deck = Deck.getInstance();
```

3.0 Thread Synchronization Mechanisms

The playTurn() method in the GameController class is vital for orderly player turn. It synchronizes access to the current player's turn, preventing conflicts. It uses playerQueue to manage turns sequentially. Also, it checks for game-over conditions. This method ensures synchronized and glitch-free gameplay.

Moreover, the implementation of locks and conditions adds a layer of sophistication to the thread synchronization mechanism. The use of the lock object, combined with the wait() and notifyAll() methods, enables controlled thread coordination. Threads that need to wait for a specific condition can efficiently do so using the lock's wait() method, releasing the lock temporarily and allowing other threads to proceed. Subsequently, the notifyAll() method awakens waiting threads when conditions change, avoiding unnecessary resource contention and ensuring synchronized progression.

By employing these synchronization mechanisms, the code optimizes resource utilization and prevents data inconsistency and deadlocks that might occur, making it suitable for managing complex interactions in a concurrent gameplay environment. This approach promotes efficient use of computational resources while maintaining data accuracy and integrity across various player actions and interactions.

4.0 Clean code principles

4.1 Meaningful and consistent names

Used meaningful and descriptive names in all parts of the code that represents the functionality, and it follows consistent naming convention and formatting style of it such as:

1. Variables

```
System.out.println("\nDiscarding cards before starting the game...\n");
int numberOfPlayers = playerQueue.size();
int cardsPerPlayer = deck.getCards().size() / numberOfPlayers;
int extraCards = deck.getCards().size() % numberOfPlayers;
```

Here I calculated a way to know the cars per play for what the size is and how many extra cards to add and to who

2. Methods

```
private void initializeCards(){
    String[] suits = {"♠", "♣", "♦", "♥"};
    String[] values = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"};
    for (String suit : suits) {
```

3. Exceptions

```
Scanner input = new Scanner(System.in);
System.out.println("Enter the number of players: ");
int numberOfPlayers = input.nextInt();
if(numberOfPlayers < 2){
    throw new InvalidInputException("You need at least 2 players to play. Try again.");
}
if(numberOfPlayers > 8){
    throw new InvalidInputException("Maximum number of players is 8. Try again.");
}
```

4.2 comments

Used comments only when needed for the parts that need to be understood.

```
Player player = playerQueue.removeCurrentPlayer();
if (player.getCardsInHand().isEmpty()) {
    System.out.println("Player " + player.getPlayerNumber() + " has discarded all their cards and
    continue; //skip empty player card
}
playerQueue.getQueue().add(player); //only add to the queue when a player can play again
```

4.3 Function Cohesion

Methods adhere to the Single Responsibility Principle (SRP), focusing on specific tasks. This practice promotes concise and modular code, facilitating easier debugging and updates.

4. 4 Don’t Repeat Yourself

Duplication is minimized through the centralization of repetitive tasks in utility classes like GameUtil. This consolidation adheres to the DRY (Don't Repeat Yourself) principle, curbing redundancy and promoting maintainability.

5.5 Whitespace and Formatting

Consistent code formatting, indentation, and spacing enhance visual clarity.